



US009411979B2

(12) **United States Patent**
Agarwal

(10) **Patent No.:** **US 9,411,979 B2**
(45) **Date of Patent:** **Aug. 9, 2016**

(54) **EMBEDDING SECRET DATA IN CODE**

(71) Applicant: **VMware, Inc.**, Palo Alto, CA (US)

(72) Inventor: **Rakesh Agarwal**, Palo Alto, CA (US)

(73) Assignee: **VMware, Inc.**, Palo Alto, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **14/500,754**

(22) Filed: **Sep. 29, 2014**

(65) **Prior Publication Data**

US 2016/0042195 A1 Feb. 11, 2016

Related U.S. Application Data

(60) Provisional application No. 62/034,450, filed on Aug. 7, 2014.

(51) **Int. Cl.**

G06F 7/04 (2006.01)
G06F 12/00 (2006.01)
G06F 17/30 (2006.01)
G06F 21/62 (2013.01)
G06F 9/445 (2006.01)
G06F 21/53 (2013.01)
G06F 9/455 (2006.01)
G06F 21/52 (2013.01)
G06F 21/12 (2013.01)

(52) **U.S. Cl.**

CPC **G06F 21/6218** (2013.01); **G06F 9/44505** (2013.01); **G06F 21/53** (2013.01); **G06F 9/455** (2013.01); **G06F 21/12** (2013.01); **G06F 21/52** (2013.01); **G06F 21/62** (2013.01); **G06F 21/6281** (2013.01)

(58) **Field of Classification Search**

CPC . G06F 21/62; G06F 21/6218; G06F 21/6281; G06F 9/44505; G06F 9/455; G06F 9/45504; G06F 21/12; G06F 21/52; G06F 21/53
USPC 726/17, 26, 27
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,809,160 A 2/1989 Mahon et al.
6,463,535 B1 10/2002 Drews
6,704,873 B1 * 3/2004 Underwood H04L 63/02 709/223
7,757,295 B1 7/2010 Hadaaegh

(Continued)

OTHER PUBLICATIONS

Office Action dated Feb. 16, 2016 in related U.S. Appl. No. 14/550,881.

(Continued)

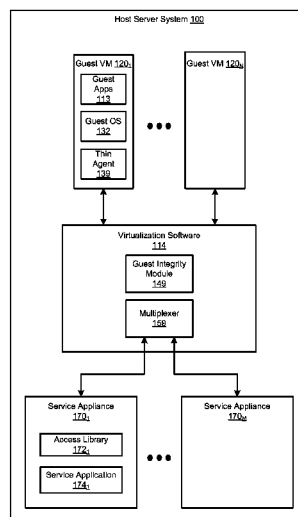
Primary Examiner — Jayesh Jhaveri

(57)

ABSTRACT

In a computer system operable at more than one privilege level, an application is securely customized to use secret data without disclosing the secret data to a managing operating system. In operation, an integrity module executes at a higher privilege level than both the managing operating system and the application. After the managing operating system loads the application executable code, the integrity module injects the secret data directly into the instruction stream of the application executable code and then sets the memory location of the secret data as executable-only. As the application executes at the assigned privilege level, the instruction in the application directly accesses the secret data without performing any indirect memory access, thereby protecting the secret data from malicious attempts to read the secret data at a privilege level lower than the integrity module.

13 Claims, 4 Drawing Sheets



Page 2

* cited by examiner

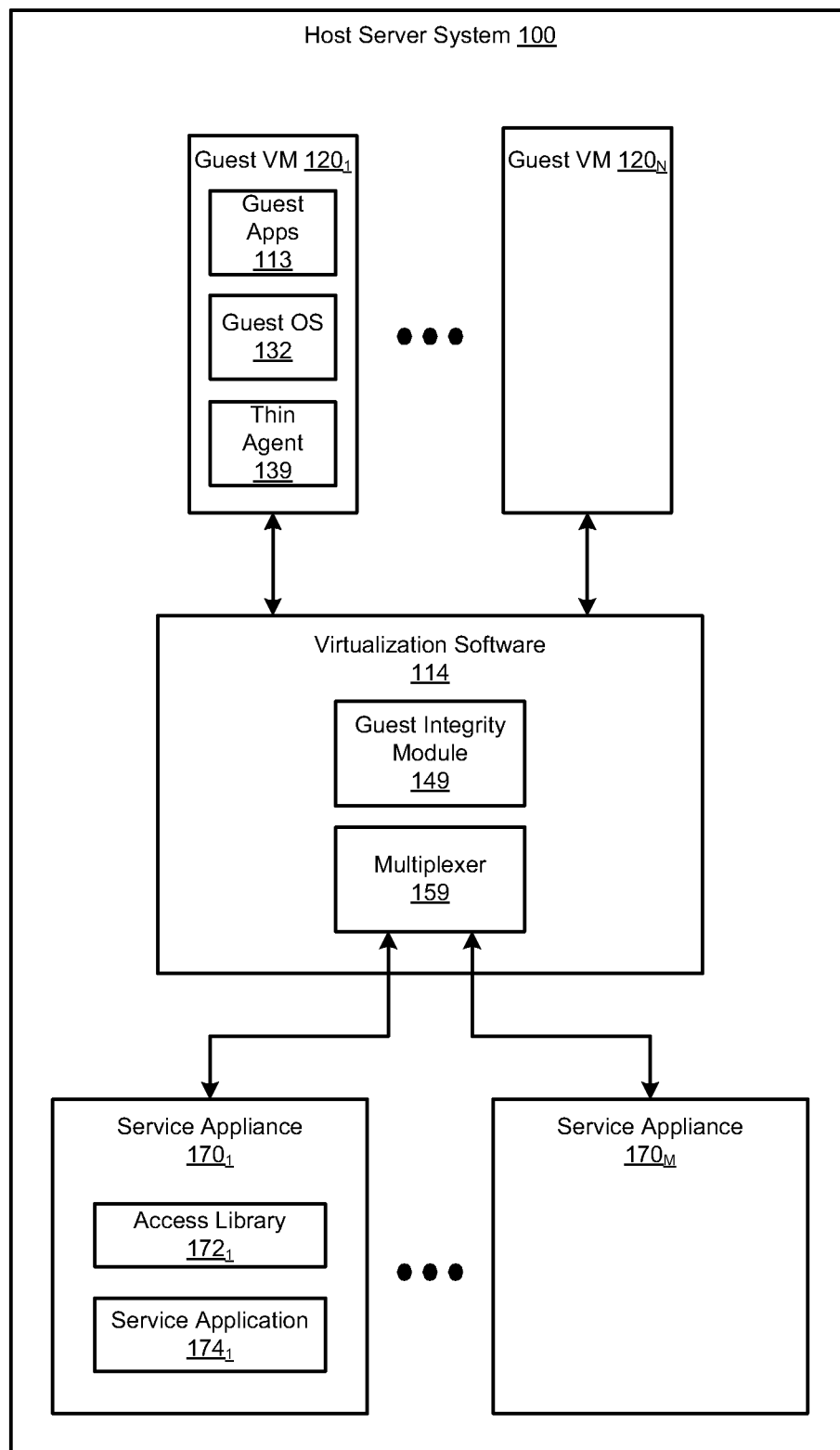


FIGURE 1

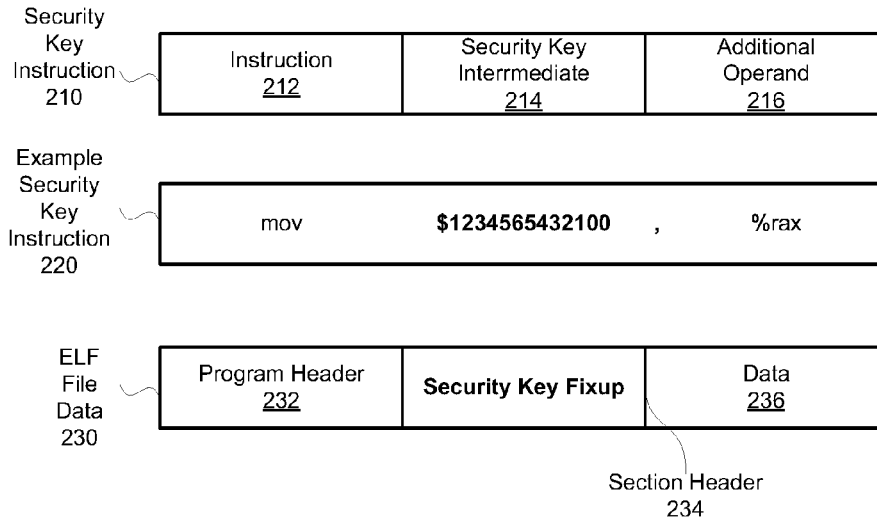


FIGURE 2A

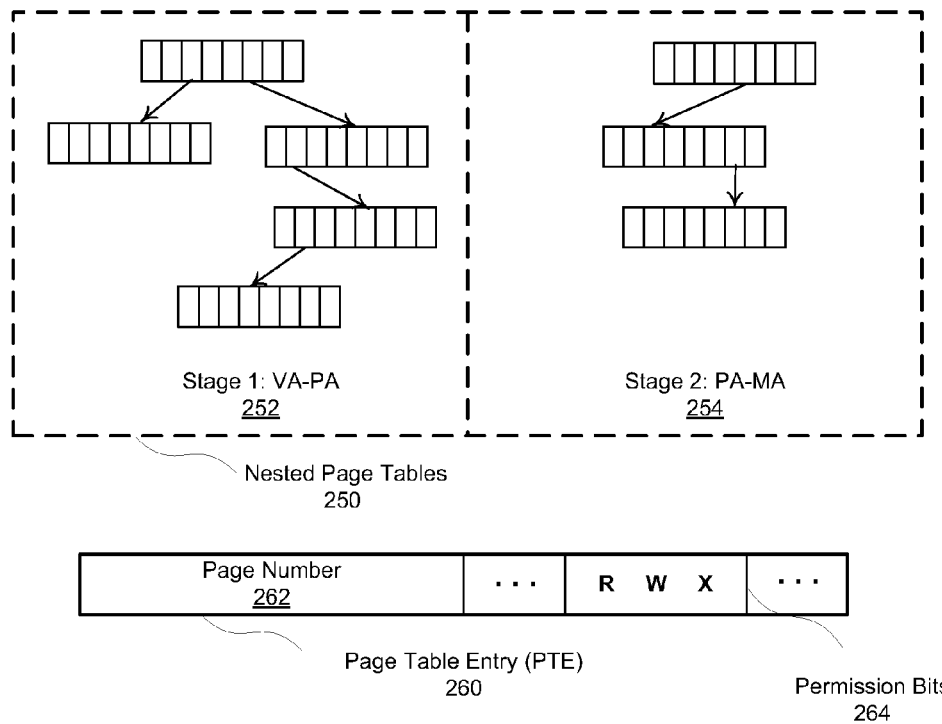
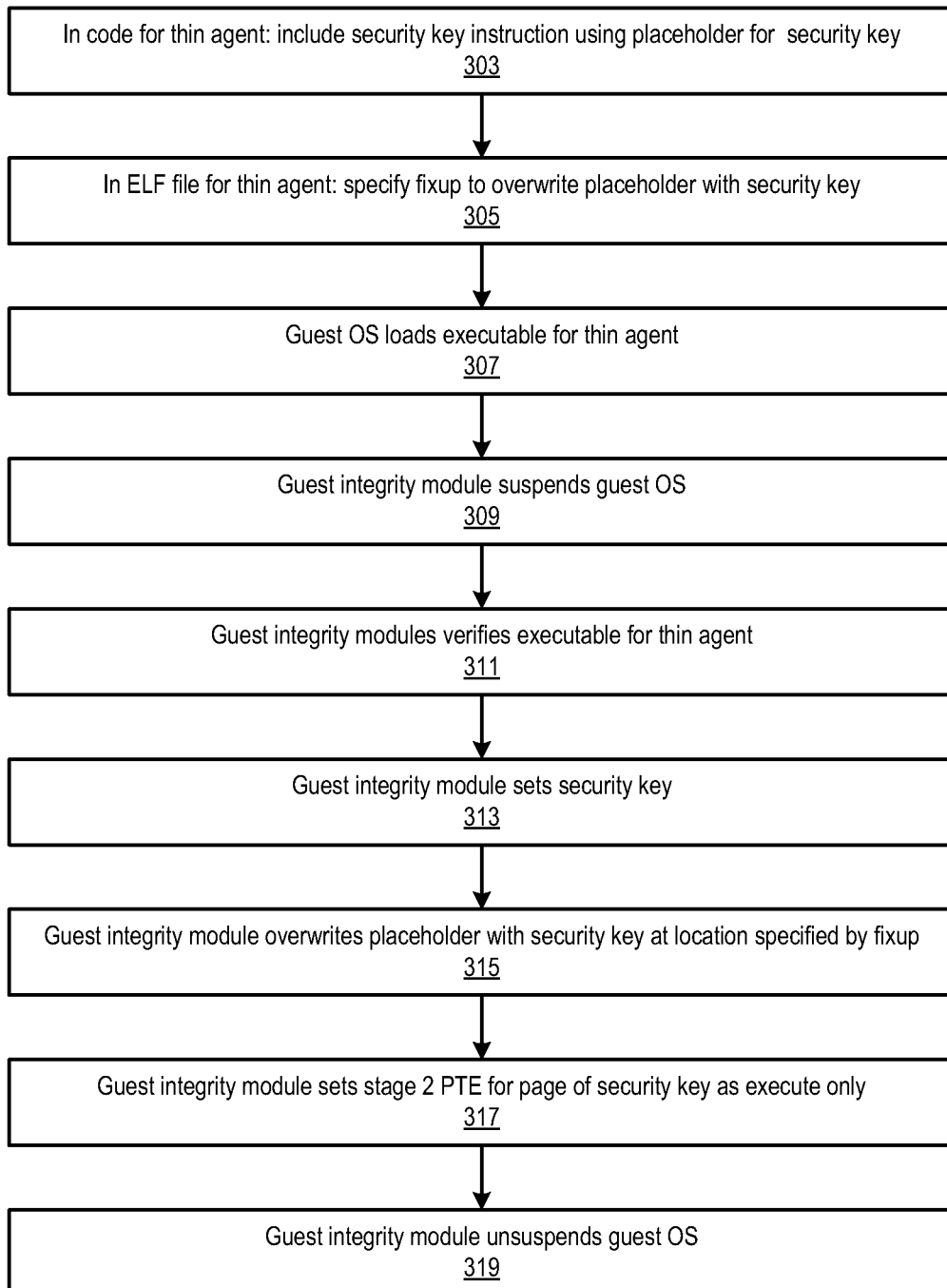


FIGURE 2B

**FIGURE 3**

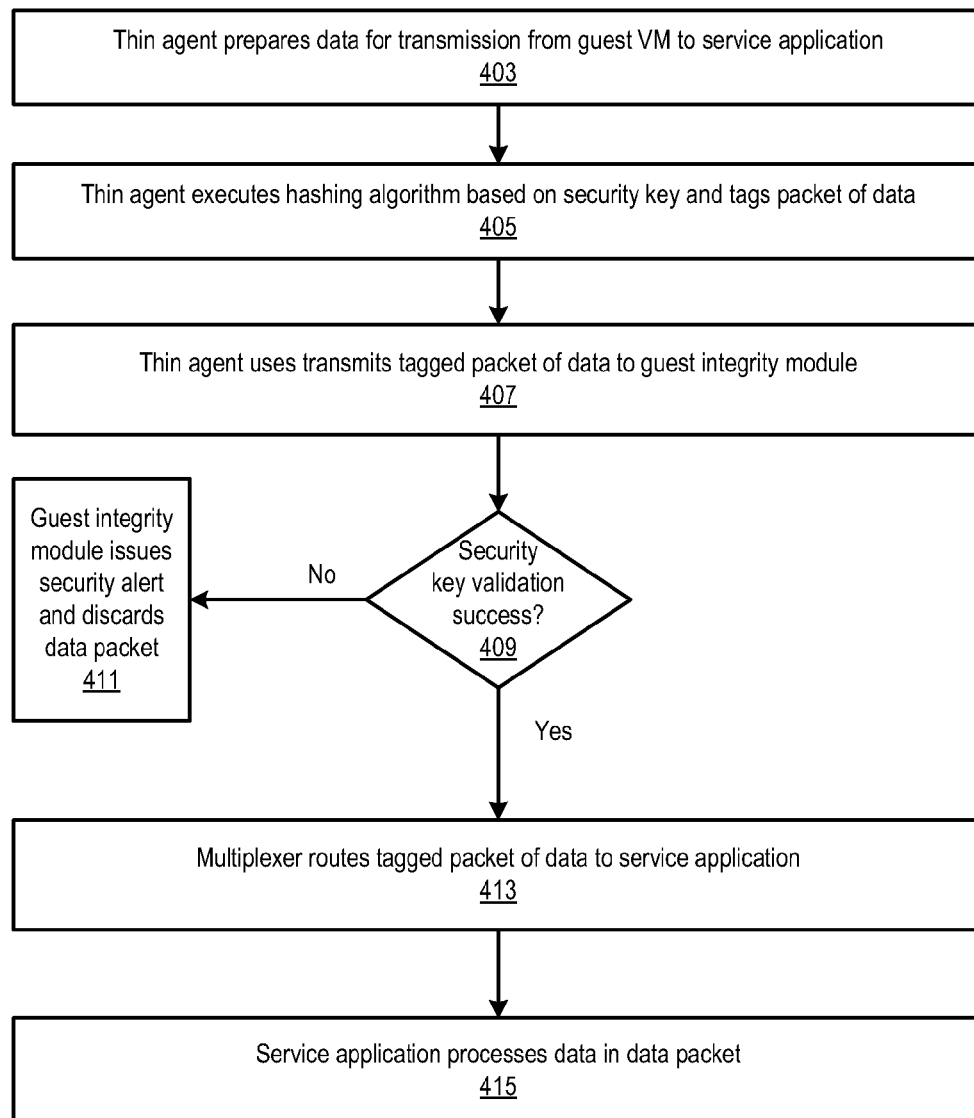


FIGURE 4

1

EMBEDDING SECRET DATA IN CODE**CROSS-REFERENCE TO RELATED APPLICATION(S)**

This application claims the benefit of U.S. Provisional Application No. 62/034,450, filed Aug. 7, 2014.

BACKGROUND

Computer systems and applications executing within the computer systems are often configured to implement security measures designed to thwart malicious activity, such as corrupting memory or accessing privileged information. For example, two separate authorized software applications may share a confidential passkey that the applications use to authenticate data transmissions. However, while such an authentication process reduces the security risk associated with a malicious user intercepting data transmissions, the security provided may be breached if the confidential pass key is obtained by the malicious user.

As illustrated by this example of data transmissions secured by passkey, maintaining the secrecy of some amount of data that is used by software applications is useful for implementing comprehensive security measures. In one attempt to limit the exposure of confidential data to unauthorized access, the confidential data is changed frequently. While this approach limits the vulnerability of the confidential data to the change interval, such an approach does not eliminate the vulnerability of the confidential data. Further, in some scenarios, the flexibility required to change confidential data periodically is not available or is too expensive to implement. Consequently, there is a need for securing confidential data used by software applications, particularly in the presence of a potentially compromised operating system, in a more effective manner.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a virtualized host server system that enables secure communications between a guest virtual machine and a service appliance.

FIGS. 2A and 2B are conceptual diagrams that illustrate how guest integrity module injects secure data into executable code.

FIG. 3 depicts a flow diagram of method steps for injecting secret data into executable code of an application.

FIG. 4 depicts a flow diagram of method steps for authenticating a data packet using secret data embedded in executable code.

DETAILED DESCRIPTION

FIG. 1 is a block diagram of a virtualized host server system 100 that enables secure communications between a guest virtual machine 120 and a service appliance 170. Host server system 100 is built on an underlying hardware computing platform comprising one or more computer systems, each of which may be a desktop computer, laptop computer, tablet computer, mobile device such as a smart phone, server grade computer system, or any other suitable hardware computing platform, including systems based on different variations of the well-known ARM or x86 architecture platforms. Host server system 100 is configured to execute virtualization software 114, one or more guest virtual machines (VMs) 120, and one or more service appliances 170.

2

Each guest VM 120 is configured to execute a guest operating system (OS) 132, which may be a commodity operating system, such as Microsoft Windows® operating system or Linux® operating system. Each guest VM 120 is further configured to support guest applications (apps) 113 and a thin agent 139. In one embodiment, thin agent 139 is an in-guest driver that executes as part of guest OS 132 and provides an interface, including communications and access for certain system operations, to a service application 174 that executes as part of service appliance 170. In alternate embodiments, thin agent 139 is any in-guest (i.e., executes in guest OS 132) component of any application that executes primarily outside guest OS 132.

Virtualization software 114 is configured to manage and operate host server system 100. Virtualization software 114 provides an execution environment for guest VMs 120 and service appliances 170. Each guest VM 120 and service appliance 170 executes as an application in an independent context, and virtualization software 114 provides a more privileged context that may be used as a bridge between these independent contexts. Virtualization software 114 may be implemented to include a kernel with hardware drivers for managing related hardware subsystems within host server system 100. In one embodiment, virtualization software 114 comprises a host operating system configured to provide system services to guest VMs 120. In other embodiments, virtualization software 114 comprises a hypervisor configured to provide certain system services to guest VMs 120. The hardware subsystems may include, without limitation, computational resources, mass storage, a networking interface, input/output interfaces, a display controller, and power management functions.

As shown, virtualization software 114 includes a multiplexer 159 and a guest integrity module 149 that both operate in the privileged context of virtualization software 114. Among other things, guest integrity module 149 works together with multiplexer 159 to forward data messages between at least one thin agent 139 and at least one service appliance 170. In one embodiment, multiplexer 159 implements a forwarding table that includes at least one entry for each thin agent 139 and each service appliance 170. In such an embodiment, multiplexer 159 implements destination based forwarding, whereby a data message is constructed to include a destination address that corresponds to at least one thin agent 139 or at least one service appliance 170. When multiplexer 159 receives the data message, an associated destination address is matched to an entry within the forwarding table to determine a destination thin agent 139 or service appliance 170 for the data message. The destination thin agent 139 or service appliance 170 may be identified using a TCP/IP (transport control protocol/internet protocol) address, a socket number, a VM identifier, or any other technically feasible identifier.

Each service appliance 170 includes software service application 174 and an access library 172. A given service appliance 170 may execute as an application under control of virtualization software 114, and may be implemented as a virtual machine with a guest OS that is configured to execute service application 174. In some embodiments, service applications 174 that implement security services may execute as applications under the control of virtualization software 114, and are implemented in a single virtual machine, known as a "security virtual machine." Access library 172 is configured so as to communicate with at least one thin agent 139 via the multiplexer 159. In one embodiment, access library 172 opens a different socket connection, for example via TCP/IP, to multiplexer 159 for communication with each different

thin agent 139. In alternative embodiments, different message passing techniques may be implemented. For example, a shared memory message passing system may be implemented for communication between thin agents 139 and access libraries 172. In certain embodiments, service appliance 170_M is configured to execute on a remote host server system that is coupled to host server system 100 via a data network. In such embodiments, service appliance 170_M establishes data connections, such as TCP/IP connections, to one or more guest VMs 120 within host server system 100 and operates substantially identically to other service appliances 170. Similarly, service appliance 170_I, executing within host server system 100, may connect to and provide services to VMs operating within the remote host server system.

Access library 172 presents an application programming interface (API) (not shown) to service application 174. The API includes service calls for communicating with at least one thin agent 139. Communicating may include, without limitation, establishing a connection with thin agent 139, configuring thin agent 139, receiving event alerts from thin agent 139, and accessing system resources for guest OS 132 associated with thin agent 139. Events that may be reported include file system events, process events, memory events, registry events, and user events. Exemplary file system events include opening a file, closing a file, writing a file, and modifying a file. Exemplary process scheduling events include mapping a file for execution, starting a process, and stopping a process. Certain types of events, such as registry events, may depend on a particular version of guest OS 132. The API may specify that certain events not be reported. For example, service application 174 may request that no events be reported, or that only specific events be reported. In one embodiment, the API enables a connection from service application 174 to a specified thin agent 139 within a guest VM 120 to be established through multiplexer 159.

In this fashion, access library 172 and thin agent 139 operate in concert to provide service application 174 with access to system resources for associated guest OS 132. However, since thin agent 139 executes as part of guest OS 132, thin agent 139 is susceptible to any security breach that compromises the integrity of guest OS 132. For example, in some embodiments, thin agent 139 and service application 174 share a security key to authenticate data packets that are communicated between thin agent 139 and service application 174 via multiplexer 159 and access library 172. In such embodiments, the security of the communications is limited by the confidentiality of the security key. For this reason, embodiments provide a guest integrity module 149 that executes within virtualization software 114 and is programmed to confidentially inject the security key provided by service application 174 into thin agent 139 prior to the transmission of data packets between thin agent 139 and service application 174. Because guest integrity module 149 operates at a more privileged security level than guest OS 132, guest integrity module 149 is able to bridge the context gap between thin agent 139 and service application 174 without exposing service application 174 to a potentially compromised guest OS 132.

It should be recognized that the various terms, layers and categorizations used to describe the virtualization components in FIG. 1 may be referred to differently without departing from their functionality or the spirit or scope of the invention. For example, host server system 100 may include virtual machine monitors (VMM) (not shown) which implement the virtual system support needed to coordinate operations between virtualization software 114 and their respective VMs. One example of virtualization software 114 that may be

used is a hypervisor included as a component of VMware's vSphere® product, which is commercially available from VMware, Inc. of Palo Alto, Calif. It should further be recognized that other virtualized computer systems are contemplated, such as hosted virtual machine systems, where the hypervisor is implemented in conjunction with a host operating system.

FIGS. 2A and 2B are conceptual diagrams that illustrate how guest integrity module 149 injects secure data into executable code. In general, guest integrity module 149 exploits the capabilities of host server system 100, such as multiple security levels, to allow guest integrity module 149 to inject secret data into code that executes on guest OS 132 while preventing guest OS 132 from reading the embedded secret data. In one embodiment, guest integrity module 149 uses a security key instruction 210 in the executable code of thin agent 139 to hold secure data, ELF file data 230 to specify the address at which to insert secure data, and nested page tables 250 to disable read and write access to the secure data. Guest integrity module 149 coordinates these activities to maintain confidentially both while guest integrity module 149 injects secret data into thin agent 139 as well as while guest OS 132 executes thin agent 139.

As shown, security key instruction 210 encapsulates a security key intermediate 214, which is a constant operand. When security key instruction 210 is executed by guest OS 132, an instruction 212 is performed on security key intermediate 214 and any additional operands 216. Because security key intermediate 214 is an "immediate" operand, the value of security key intermediate 214 is directly available in the instruction stream, and to security key instruction 210. By contrast, one or more additional operands 216 may specify memory addresses that indirectly reference the corresponding values. It should be recognized that specifying the value of a constant via security key intermediate 214 insulates the constant from security risks associated with memory accessible to guest OS 132.

An example security key instruction 220 illustrates one instruction that is used to protect secret data. As shown, security key intermediate 214 is the value "1234565432100" where the immediate type is designated by "\$," instruction 212 is "mov," and additional operand 216 is "%rax" (i.e., the rax register). When guest OS 132 executes example security key instruction 220, the processing unit loads secret data "1234565432100" into the rax register from where it can be manipulated further, such as being incorporated into a hashing algorithm.

To effectively inject secret data into executable code requires modifying the executable code in a confidential environment. Since guest integrity module 149 executes at a higher privilege level than guest OS 132, guest integrity module 149 is able to inject the secret data without disclosing the secret data to guest OS 132. In one embodiment, guest integrity module 149 receives the secret data from service application 174 via a secure transmission. Subsequently, guest integrity module 149 accesses Executable and Linkable format (ELF) file data 230 that includes the executable code in addition to one or more relocation entries, known as "fixups," that specify addresses in the executable code that are to be set to the secret data after the executable code is loaded.

As shown, ELF file data 230 includes a program header 232, a section header 234, and data 236. Notably, section header 234 includes "security key fixup" that specifies the address of security key intermediate 214. In operation, as part of preparing executable code that is intended to hold the security key, the software developer of the executable code inserts security key fixups into the corresponding ELF data

file **230**—conveying that guest integrity module **149** is to overwrite security key intermediates **214** with the security key.

In alternate embodiments, ELF may be replaced with any other format that supports relocations, such as Windows® Preinstallation Environment. Further, it should be recognized that other techniques for obtaining and injecting secret data into executable code may be employed by guest integrity module **149**.

To continuously hide the secret data from read access by guest OS **132**, guest integrity module **149** uses the addressing and protection mechanisms provided by host server system **100**. Host server system **100** carries out mappings from a guest virtual address space of guest VMs **120₁-120_N** or any other applications running virtualization software **114** to a machine address space of memory (referred to herein as the “host physical address space”) using nested page tables **250**. As shown in FIG. 2B, nested page tables **250** is a page translation hierarchy that includes a stage 1: guest virtual address (VA) to guest physical address (PA) **252** and a stage 2: PA to machine address (MA) **254**.

Both stage 1: VA-PA **252** and stage 2: PA-MA **254** include page table entries (PTEs) **260**, and each PTE **260** includes, inter alia, a page number **262** and permission bits **264**. Page number **262** indicates the next page in the page table translation hierarchy. If PTE **240** is at the lowest level of the page table translation hierarchy, then page number **262** corresponds to a data page. In general, attributes, such as permission bits **264**, associated with a data page are defined by the more restrictive of the stage 1: VA-PA **252** and the stage 2: PA-MA **254** attributes traversed in the page table translation hierarchy. As shown, permission bits **264** include a read “R” bit, a write “W” bit, and execute “X” bit. For PTE **260**, if the read and write bits are each clear and the execute bit is set, then the instruction referenced by PTE **260** is execute-only. If any PTE **260** along the page table translation hierarchy traversed to translate an address in guest virtual address space to host physical address space is designated as execute-only, then guest OS **132** is unable to read from the address or write to the address, but is able to execute an instruction at the address.

In host server system **100**, guest OS **132** is capable of modifying PTEs **260** included in stage 1: VA-PA **252** either via software or hardware mechanisms, but guest OS **132** is unable to modify PTEs **260** included in stage 2: PA-MA **254**. By contrast, guest integrity module **149** is capable of modifying PTEs **260** included in stage 2: PA-MA **254**. Guest integrity module **149** leverages this difference in PTE **260** accessibility in conjunction with permissions bits **264** to protect security key instruction **210** from attempts to ascertain the value of security key intermediate **214** by guest OS **132**.

More specifically, before performing the security key fixup specified in ELF file data **230**, guest integrity module **149** suspends guest OS **132** and enables write access to PTE **260** corresponding to security key instruction **210**. After performing the security key fixup, guest integrity module **149** disables read and write access to PTE **260** corresponding to security key instruction **210**, enables execute-only access to PTE **260**, and unsuspends guest OS **132**. Since guest OS **132** is unable to read security key intermediate **214** included in security key instruction **210**, the value of security key intermediate **214** is protected from security breaches that compromise guest OS **132**, guest apps **113**, and thin agent **139**.

It should be understood that FIG. 2B illustrate one possible configuration of a page table translation hierarchy—nested page tables **250**—and bits in PTE **260**, and the number and arrangement of elements in the page table translation hierar-

chy and PTE **260** can be varied from what is shown. Host server system **100** may employ any number of translation lookaside buffers (TLBs) as part of the page translation process. Further, in alternate embodiments, host server system **100** may carry out mappings from the guest virtual address space to the host physical address space using shadow page tables to map guest virtual address spaces within guest VMs **120₁-120_N** directly to the physical address space of memory. Embodiments include any mechanism that enables guest integrity module **149** and not guest OS **132** to designate the address corresponding to security key intermediate **214** or a range of addresses that includes the address corresponding to security key intermediate **214** as execute-only. For example, in some architectures, the code pages accessed via extended page tables may be “tagged” as execute-only.

FIG. 3 depicts a flow diagram of method steps for injecting secret data into executable code of an application. In the embodiment illustrated herein, thin agent **139** is an in-guest driver that executes as part of guest OS **132** and provides an interface, including communications and access for certain system operations, to service application **174**. Service application **174** has established a security key which is unknown to guest OS **132**. Although guest OS **132** may be compromised, service appliance **170** and virtualization software **114** execute in different contexts and the integrity of these contexts is unaffected by compromised guest OS **132**.

This method begins at step **303** where a software developer includes security key instruction **210** in the code of thin agent **139**. As part of security key instruction **210**, the software developer includes a placeholder (e.g., a meaningless value to be replaced later) as security key intermediate **214**. At step **305**, the software developer inserts a security key fixup into ELF data file **230** corresponding to thin agent **139**. The security key fixup specifies the address of the placeholder and conveys that this address is to be overwritten with the security key.

At step **307**, guest OS **132** loads ELF file data **230** corresponding to thin agent **139**. More specifically, guest OS **132** loads the executable for thin agent **139** and may perform any number of fixups that are not associated with the security key. At step **309**, guest integrity module **149** suspends guest OS **132**. While guest OS **132** is suspended, guest OS **132** is unable to perform operations and, consequently, guest integrity module **149** may alter page table entries **260** and the executable of thin agent **139** without detection by guest OS **132**. However, if guest OS **132** is already compromised, then the executable of thin agent **139** may also be compromised. To thwart any attempt at breaching security via thin agent **139**, guest integrity module **149** verifies the executable for thin agent **139** before exposing thin agent **139** to any secure data. Guest integrity module **149** may verify the executable for thin agent **139** in any technically feasible fashion. For instance, guest integrity module **149** may independently load a second copy of the executable of thin agent **139** via ELF file data **230** and then compare the two copies of the executable of thin agent **139**.

At step **313**, guest integrity module **149** establishes the security key to match the security key that is expected by service application **174**. For example, if service application **174** incorporates the security key as part of a hashing algorithm, then security key instruction **210** may be configured to cause thin agent **139** to use the same security key as part of a complementary hashing algorithm. Guest integrity module **149** may establish this security key in any fashion that preserves the confidentiality of the security key. For instance, guest integrity module **149** may receive the security key in a confidential transmission from service application **174** and

then ensure that the security key is not stored in memory accessible to guest OS 132 at any time guest OS 132 is active (i.e., unsuspended). After establishing the security key, guest integrity module 149 processes the security key fixup in ELF file data 230, overwriting the placeholder that is located at the address of security key intermediate 214 with the security key.

To ensure the continued confidentiality of the security key, while guest OS 132 is still suspended, guest integrity module 149 modifies permission bits 264 in stage 2: PA-MA 254 page table entry 260 corresponding to the page that includes the address of security key intermediate 214. More specifically, guest integrity module 149 disables read and write access, and enable execute-only access for a range of addresses that includes security key instruction 210 (step 317). At step 319, guest integrity module 149 unsuspends guest OS 132, and thin agent 139 executes in guest OS 132. In operation, thin agent 139 and service application 174 cooperatively use the security key embedded in thin agent 139 and known to service application 174 for validation purposes, without exposing the security key to security risks of guest OS 132.

FIG. 4 depicts a flow diagram of method steps for authenticating a data packet using secret data embedded in executable code. In the embodiment illustrated herein, thin agent 139 executes as part of guest OS 132 in guest VM 120 context and service application 174 executes as part of service appliance 170 in a separate, service appliance 170 context, such as a security VM. Guest integrity module 149 executes as part of virtualization software 114 in a context that executes at a more privileged permission level than guest VM 120 context, enabling guest integrity module 149 to act as a secure bridge between guest VM 120 and service appliance 170 contexts. The code of thin agent 139 includes security key instruction 210 in which security key intermediate 214 is set to a security key known to thin agent 139, guest integrity module 149, and service application 174, but not to guest OS 132. Further, security key intermediate 214 is designated as execute-only and, consequently, thin agent 139 may execute security key instruction 210 without exposing the value of the security key to guest OS 132.

This method begins at step 403 where thin agent 139 prepares data for transmission from guest VM 120 to service appliance 170. As part of preparing the data, thin agent 139 executes a hashing algorithm that includes security key instruction 210 (step 405). Since security key instruction 210 includes the value of the security key as security key intermediate 214, thin agent 139 is able to execute the hashing algorithm without loading the security key in memory accessible by guest OS 132, and thus preserves the confidentiality of the security key. Thin agent 139 then tags the packet of data based on the hashing algorithm, and transmits the tagged packet of data to guest integrity module 149 (step 407) in any technically feasible fashion. In some embodiments, thin agent 139 accesses an I/O port specific to guest VMs 120, known as a backdoor port, to communicate with virtualization software 114, including guest integrity module 149.

Upon receiving the tagged packet of data, guest integrity module 149 performs verification operations to ensure that the security key is valid. If, at step 409, guest integrity module 149 determines that the security key is not valid, then guest integrity module 149 issues a security alert and discards the data packet 411. In this fashion, if guest OS 132 is compromised and attempts to transmit malicious data to service application 174, then the security key enables guest integrity module 149 to intercept the data, isolating service application 174 from compromised guest OS 132.

At step 409, if guest integrity module 149 determines that the security key is valid, then guest integrity module 149 forwards the tagged data packet to multiplexer 159. At step 413, multiplexer 159 routes the tagged data packet to service application 174. At step 415, service application 174 receives the tagged data packet and processes the data in the data packet. As part of receiving the tagged data packet, service application 174 may execute a hashing algorithm that is complementary to the hashing algorithm implemented by thin agent 139—using the shared, but confidential, security key to recreate the original data.

The embodiments disclosed herein detail specific use cases in which embedding secure data in code executed by a compromised operating system effectively thwarts attempts by the compromised operating system to obtain the secure data. However, the described embodiments are to be considered as illustrative and not restrictive. Other use cases that leverage the same underlying techniques to securely inject secret data in code are envisioned. Further, although host server system 100 supports a virtualization environment, any host server system that supports at least two levels of privilege and provides a mechanism to designate certain addresses as execute-only may implement these security techniques.

The various embodiments described herein may employ various computer-implemented operations involving data stored in computer systems. For example, these operations may require physical manipulation of physical quantities—usually, though not necessarily, these quantities may take the form of electrical or magnetic signals, where they or representations of them are capable of being stored, transferred, combined, compared, or otherwise manipulated. Further, such manipulations are often referred to in terms, such as producing, identifying, determining, or comparing. Any operations described herein that form part of one or more embodiments of the invention may be useful machine operations. In addition, one or more embodiments of the invention also relate to a device or an apparatus for performing these operations. The apparatus may be specially constructed for specific required purposes, or it may be a general purpose computer selectively activated or configured by a computer program stored in the computer. In particular, various general purpose machines may be used with computer programs written in accordance with the teachings herein, or it may be more convenient to construct a more specialized apparatus to perform the required operations.

The various embodiments described herein may be practiced with other computer system configurations including hand-held devices, microprocessor systems, microprocessor-based or programmable consumer electronics, minicomputers, mainframe computers, and the like.

One or more embodiments of the present invention may be implemented as one or more computer programs or as one or more computer program modules embodied in one or more computer readable media. The term computer readable medium refers to any data storage device that can store data which can thereafter be input to a computer system—computer readable media may be based on any existing or subsequently developed technology for embodying computer programs in a manner that enables them to be read by a computer. Examples of a computer readable medium include a hard drive, network attached storage (NAS), read-only memory, random-access memory (e.g., a flash memory device), a CD (Compact Discs)—CD-ROM, a CD-R, or a CD-RW, a DVD (Digital Versatile Disc), a magnetic tape, and other optical and non-optical data storage devices. The computer readable medium can also be distributed over a network coupled com-

puter system so that the computer readable code is stored and executed in a distributed fashion.

Although one or more embodiments of the present invention have been described in some detail for clarity of understanding, it will be apparent that certain changes and modifications may be made within the scope of the claims. Accordingly, the described embodiments are to be considered as illustrative and not restrictive, and the scope of the claims is not to be limited to details given herein, but may be modified within the scope and equivalents of the claims. In the claims, elements and/or steps do not imply any particular order of operation, unless explicitly stated in the claims.

Virtualization systems in accordance with the various embodiments may be implemented as hosted embodiments, non-hosted embodiments or as embodiments that tend to blur distinctions between the two, are all envisioned. Furthermore, various virtualization operations may be wholly or partially implemented in hardware. For example, a hardware implementation may employ a look-up table for modification of storage access requests to secure non-disk data.

Many variations, modifications, additions, and improvements are possible, regardless the degree of virtualization. The virtualization software can therefore include components of a host, console, or guest operating system that performs virtualization functions. Plural instances may be provided for components, operations or structures described herein as a single instance. Finally, boundaries between various components, operations and data stores are somewhat arbitrary, and particular operations are illustrated in the context of specific illustrative configurations. Other allocations of functionality are envisioned and may fall within the scope of the invention(s). In general, structures and functionality presented as separate components in exemplary configurations may be implemented as a combined structure or component. Similarly, structures and functionality presented as a single component may be implemented as separate components. These and other variations, modifications, additions, and improvements may fall within the scope of the appended claim(s).

I claim:

1. A method of securely injecting data in a computer system operable in a plurality of privilege levels, comprising:

while operating at a first privilege level within a context of a virtual machine:

launching a guest operating system, and
causing the guest operating system to load executable code having a placeholder included in a processor level instruction in the executable code; and

while operating at a second privilege level within a context of a hypervisor, the second privilege level being more secure than the first privilege level:

suspending the guest operating system,
replacing the placeholder with a confidential value,
setting the instruction to be execute-only, such that the confidential value in the instruction cannot be read or overwritten by the guest operating system, and
unsuspending the guest operating system.

2. The method of claim 1, wherein setting the instruction to be execute-only comprises setting a page table entry that control access to the instruction such that read access is disabled, write access is disabled, and execution access is enabled.

3. The method of claim 1, wherein the placeholder is an immediate operand used directly by the instruction and is not a memory address.

4. The method of claim 1, wherein replacing the placeholder comprises:

identifying a location specified by a fixup, the fixup being included in the executable code; and
overwriting the placeholder at the location, specified by the fixup, with the confidential value.

5. A computer system operable in a plurality of privilege levels, comprising:

a processor; and

a memory coupled to the processor and having loaded therein a guest operating system that executes at a first privilege level and loads executable code having a placeholder included in a processor level instruction in the executable code, and a guest integrity module that operates at a second privilege level that is more secure than the first privilege level and executes a program that securely injects data in the computer system, the program comprising:

suspending the guest operating system,

replacing the placeholder with a confidential value,

setting the instruction to be execute-only, such that the confidential value in the instruction cannot be read or overwritten by the guest operating system, and
unsuspending the guest operating system.

6. The computer system of claim 5, wherein the memory further includes a hypervisor that supports execution of a virtual machine, the guest operating system executes within a context of the virtual machine, and the guest integrity module executes within a context of the hypervisor.

7. The computer system of claim 5, wherein setting the instruction to be execute-only comprises setting a page table entry that control access to the instruction such that read access is disabled, write access is disabled, and execution access is enabled.

8. The computer system of claim 5, wherein the placeholder is an immediate operand used directly by the instruction and is not a memory address.

9. The computer system of claim 5, wherein replacing the placeholder comprises:

identifying a location specified by a fixup, the fixup being included in the executable code; and
overwriting the placeholder at the location, specified by the fixup, with the confidential value.

10. A non-transitory computer-readable storage medium comprising instructions, which when executed in a computer system operable in a plurality of privilege levels and in isolated contexts, causes the computer system to carry out the steps of:

while operating at a first privilege level within a context of a virtual machine:

launching a guest operating system, and

causing the guest operating system to load executable code having a placeholder included in a processor level instruction in the executable code; and

while operating at a second privilege level within a context of a hypervisor, the second privilege level being more secure than the first privilege level:

suspending the guest operating system,

replacing the placeholder with a confidential value,

setting the instruction to be execute-only, such that the confidential value in the instruction cannot be read or overwritten by the guest operating system, and
unsuspending the guest operating system.

11. The non-transitory computer-readable storage medium of claim 10, wherein setting the instruction to be execute-only comprises setting a page table entry that control access to the instruction such that read access is disabled, write access is disabled, and execution access is enabled.

11

12. The non-transitory computer-readable storage medium of claim **10**, wherein the placeholder is an immediate operand used directly by the instruction and is not a memory address.

13. The non-transitory computer-readable storage medium of claim **10**, wherein replacing the placeholder comprises: 5
identifying a location specified by a fixup, the fixup being included in the executable code; and
overwriting the placeholder at the location, specified by the fixup, with the confidential value.

* * * * *

10

12